DTIC

S
ELECTE
NOV 07 1990
D
D^cs

An Object-Oriented UIMS
for Rapid Prototyping

TR90-016

April, 1990

N00014-86-K-0680

Yen-Ping Shan

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

A TextLab Report
UNC is an Equal Opportunity/Affirmative Action Institution.

An Object-Oriented UIMS for Rapid Prototyping

Yen-Ping Shan

Department of Computer Science, University of North Carolina
Chapel Hill, NC 27599-3175, U.S.A.
shan@cs.unc.edu
(919) 962-1874

Abstract

User interface management systems (UIMSs) that support rapid prototyping often suffer from the limited range of interfaces that they can produce and the lack of support for the connection between the produced interface and its underlying application. This paper discusses a Mode Development Environment (MoDE) that addresses these problems.

1 Introduction

Creating a good user interface for a system is a difficult task. User interface software is often large, complex, and difficult to debug and modify. It often represents a significant fraction of the code, frequently ranging from 40 to 60 percent. Good interfaces that are easy to use are also interfaces that are complex and hard to create. There are few guidelines or strategies at the design stage that will insure that the resulting user interface will be easy to learn, easy to use, and user-friendly. Instead, user interface developers

rely on testing prototypes with actual end users and iteratively modifying the design. Many user interface management systems (UIMSs) have been developed to facilitate rapid prototyping [NeX88, Sme87, LIBY89, Car89]. Although they have helped in many aspects of the prototyping process, most of them suffer from lack of generality and lack of support for connecting the interface with the application.

Many UIMSs are limited in the look and feel of the interfaces they can generate. It is very hard to generate user interfaces not in the style provided. The major reason is that they have a *fixed* library of interface components. The possible interfaces are limited to those that can be composed from components in the fixed library. For a production system, this might be desirable since it maintains consistency among the interfaces. For a prototyping system where new ideas are to be tested, lack of generality becomes a serious deficiency.

Also, few of the UIMSs for prototyping provide adequate support for connecting the user interface to the underlying application. In a good interface, the semantics of the application often strongly affect the design of the user interface. Consequently, the prototype must be connected to the application or to a model of the application if it is to be tested fully. Most UIMSs that generate a set of procedures or provide a callback mechanism require programming by the interface developer to connect the interface to the application. This programming task often becomes the bottleneck in the prototyping process.

This paper presents a Mode Development Environment (MoDE) that addresses the above problems.

2

2 MoDE

MoDE is a general user interface management system that supports rapid creation of a wide variety of user interfaces. It is implemented on top of Smalltalk-80 [GR83] and an event-driven mechanism [Sha89]. Its dynamically expandable interaction technique library allows the interface developer to easily introduce new objects into the library. MoDE also supports creation and management of the connection between the user interface and the application through direct manipulation.

An interface developer uses MoDE's library of interaction techniques to construct new interaction techniques. Each interaction technique built using MoDE may be promoted to the library for reuse at any time. The MoDE library stores the interaction techniques in the form of live objects (with values in the instance variables retained). Each library object represents a "copy," as opposed to the class, of an interaction technique. As a consequence, when promoting an interaction technique, only a live copy of the technique needs to be created and registered; there is no need to recompile the library. Furthermore, once an interaction technique is promoted into the library, it can be reused immediately by making copies of it. The above properties allow the library to be dynamically expanded. Interactive techniques stored in the library can also be written to files. These files can be read by other interface developers' libraries to share the interaction techniques.

Each interface generated by MoDE is composed of a number of basic building blocks called *modes*. A mode is distinguished by an area ou the screen that interacts differently than its surrounding areas. A user interface might be composed of a group of hierarchically structured modes. A mode in such a structured interface could contain other modes as submodes. Any given mode, however, would be a submode of only one mode – its "super-
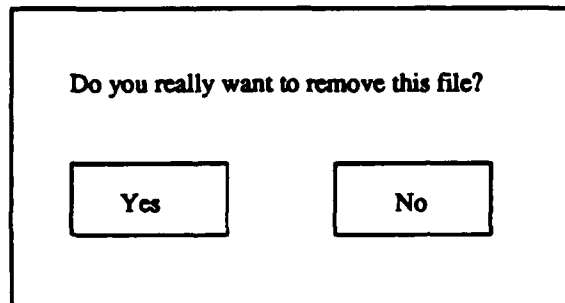
3

Figure 1: A dialogue box can be viewed as one supermode with two submodes.

mode." The set of modes in a structured interface forms a hierarchy.

To illustrate, the dialogue box shown in Figure 1 can be thought of as a mode with two submodes: a "yes" submode and a "no" submode. The yes and no buttons highlight themselves when the left mouse button is pressed within them, and they dehighlight themselves when the cursor moves away or the left mouse button is released. Their behavior is different from that of their supermode which does not respond to a left mouse button press. Notice that the text in the dialogue box is not a mode. It affects the appearance of the dialogue box, but it does not form an area that provides a different interpretation of the user's input.

Each mode has a "semantic object" that supplies its semantics. The term "supply" is used instead of "generate" because in MoDE, the actual semantics are "generated" by the application but they are "supplied" to the interface by a separate "semantic object" being described here. Semantic objects can also connect to each other. They reside in a layer maintained by MoDE. Objects in the layer have knowledge of both the user interface and the application. They insulate both sides from the effects of changes. MoDE supports the creation and manipulation of semantic objects through direct manipulation. This three-level model of interface modes, semantic objects,

and application is illustrated in the next section.

It is the existence of the semantic objects that allows MoDE to provide a rich support for the connection between the user interface and the application. Since all connections are made through the semantic objects, supporting routines can be built into the abstract superclass of the semantic objects. These routines keep track of the creation, deletion, and modification of the connections. They provide MoDE with sufficient information to perform searches, consistency checks, and other maintenance operations. Semantic objects also help in presenting the connections to the interface developer. Without them, the links between interface objects would need to be drawn directly from one to another (such as the link between the "yes" submode and the dialogue box in Figure 1). A display incorporating many such links would be difficult to understand.

3   MoDE in Use

Through a concrete example, this section illustrates how MoDE can be used to create a prototype of a simple binary desk calculator with one display window and three push buttons–"0," "1," and "C" (the clear button).

With MoDE, interfaces are created by dragging objects (modes) out of the interactive technique library (the right-hand window in Figure 2) and pasting them together. In Figure 2, the user has created a "Vanilla Mode" as the background of the calculator and is editing its appearance.

Next, the user creates the three buttons and the display window for the desk calculator and pastes them onto the background. This process is similar to drawing a picture with a drawing tool. The result is shown in Figure 3.
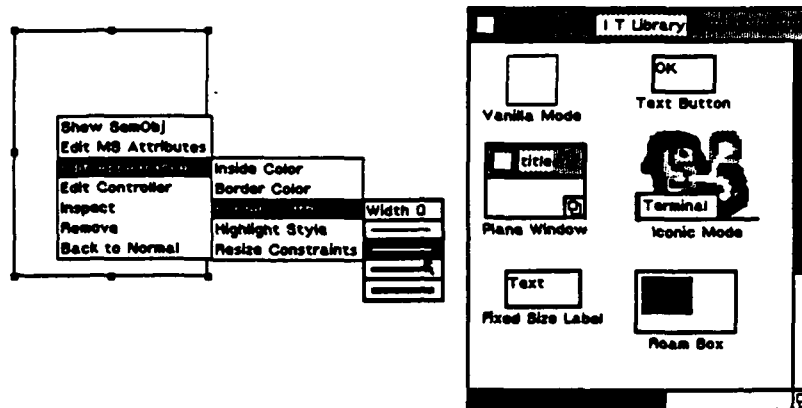
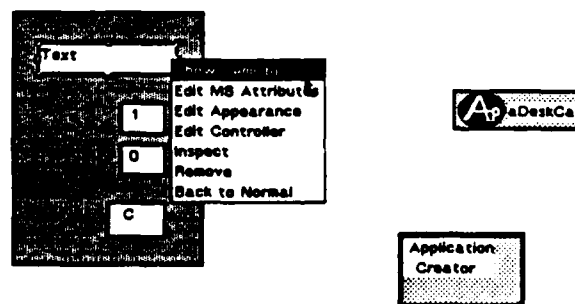Figure 2: Editing the appearance of a mode.

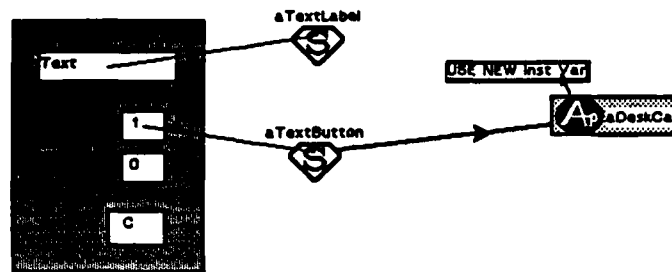Figure 3: Showing the semantic object for the display window of the desk calculator.

Figure 4: System requests permission to create new instance variable for the connection.

The "Application Creator" shown in the lower right corner of Figure 3 is used to create the representative of the computing component of the desk calculator. Because the computing component is not a visible user interface object, a visual representative is necessary for it to be displayed and manipulated directly. Here, the user decides to create the computing component from scratch. A new class named "DeskCal" is defined and an instance of the class is created. The representative of this instance (with the text "ApaDeskCal") is shown. Remember, the semantic objects are the points of connection. To establish the connection between the user interface and the computing component, the semantic objects must be present. In Figure 3 the user is requesting the system to show the representative of the semantic object of the display window.

Figure 4 shows the semantic objects (represented by diamond shaped icons containing an "S") for the display window and the "1" button. The user has created a link from the semantic object of the "1" button to the computing component, and would like to create another link from the computing component to the semantic object of the display window. His plan is for the semantic object of the "1" button to send a message to the computing component whenever the button is pushed. The computing component, in response, updates its states and requests the display window to display the
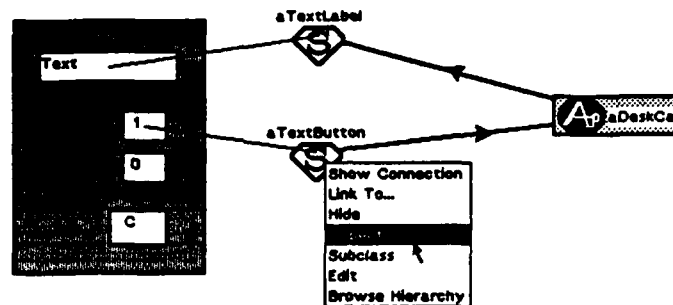
7

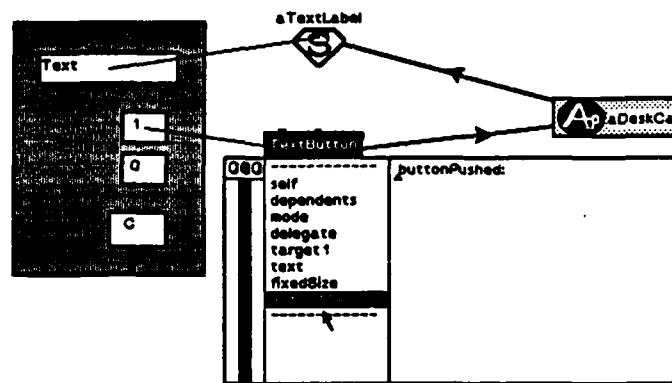Figure 5: Inspect the semantic object.

Figure 6: The default action message is "buttonPushed:"

digit "1" by sending a message to its semantic object. Since the *DeskCal* class is a new class, it does not have an instance variable to store the connection. The system infers that a new instance variable is needed and requests permission to create one, as shown in Figure 4. Once the permission is granted, the user will be prompted for the name of the new instance variable and the system will automatically change the class definition of the *DeskCal* to insert this new instance variable and update all the existing instances of the class.

Next, the user selects the "Inspect" option in the menu associated with the semantic object to inspect the "1" button (Figure 5). The inspector,
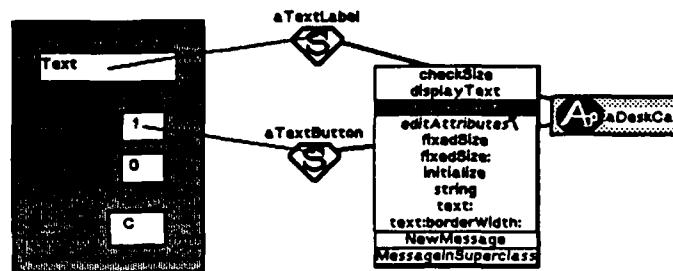
8

Figure 7: The system shows a list of the messages understood by the semantic object of the display window.

shown in Figure 6, indicates that the default action message for the button is "buttonPushed:" The colon at the end indicates that there is one argument for this message. By default it is the text string of the button.

Since the computing component is created from scratch and does not understand the "buttonPushed:" message, the user selects the "Add Message" option in the menu associated with the link. The system will open a code editor for the user to define the "buttonPushed:" method in the *DeskCal* class.

In the process of defining the method, the user needs to know what message can be sent to the display window to display the result of a computation. The system can help by displaying the messages understood by the display window. In Figure 7, the list of understood messages is shown and the user finds that the "displayText:" method is what he needs.

The other two buttons can be connected in the same manner. Figure 8 shows the fully connected desk calculator. Since all interfaces created with MoDE are immediately testable, there is no need to switch to a test state. Further, the user can test the partially implemented prototype at any point in its development. In Figure 8, for example, the button "1" was pushed and
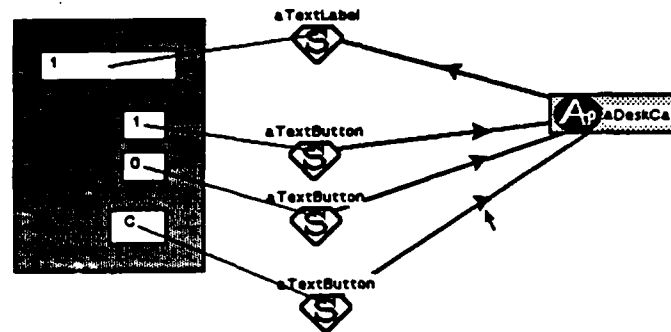
Figure 8: The interface and the application are fully connected.

the display window of the calculator shows the correct result.

There are two approachs to handling the clear button. The first one is to use the default message ("buttonPushed:") and have the computing component interpret the argument "C" as a special command. An alternative is to use a different message selector (for example "clear") and define the corresponding method in the *DeskCal* class. Both approachs are valid. MoDE allows the user to choose whichever he prefers. .

After the user finishes the prototyping, he hides all the connections and promotes the calculator into the interaction technique library by dragging the desk calculator into the library. The library automatically prepares an icon for the calculator, as shown in Figure 9.

4 Experience with MoDE

Sample Interfaces

MoDE has been used to create many direct-manipulation user interfaces. Figure 10 shows a few sample interfaces created with it. The scroll bar in the top
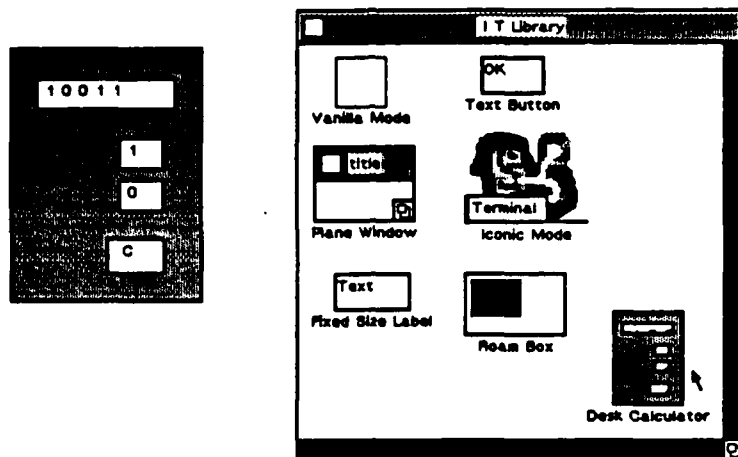
10

Figure 9: The binary desk calculator is promoted into the interaction technique library.

left window (Roam demo) scrolls the picture continuously. The top right window (Menu demo) has three types of menus: title-bar menu, tear-off menu, and pop-up menu (not displayed). Menu items can be text, foreign characters, bitmaps, and animated pictures. The lower left window (titled "For Barry") demonstrates the system's capability to incorporate scanned images and text editors. The largest window (titled "OddShape Window") contains two subwindows; both allow the user to create networks of hypertext nodes. The oddly shaped subwindow has three nodes in it. The user is dragging one of the nodes over the trash icon in another window (titled "Level of DM"). The trash icon opens to provide semantic feedback. Rubber-band lines are drawn from "Odd1" node and "Odd3" node to the node being dragged to show the connection. Notice that the oddly shaped subwindow has a hole in it through which the user can work with objects (for example, the "Below1" node) underneath the window. MoDE also supports semi-transparent windows as shown in the right half of the oddly shaped subwindow.
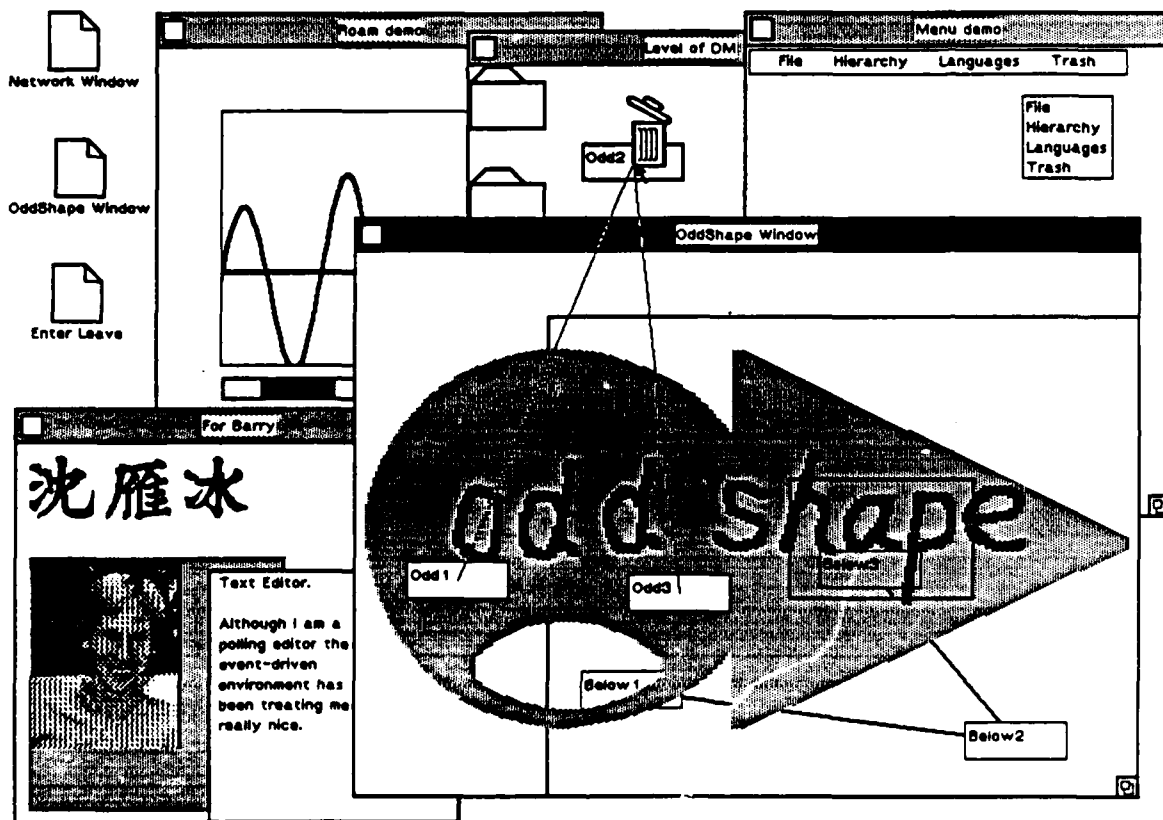
Self-Creation

11

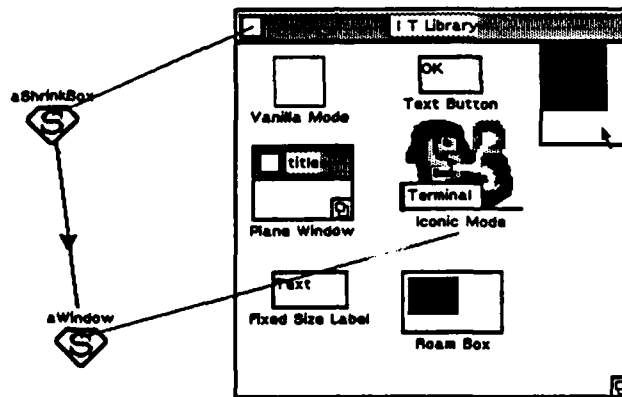Figure 10: Sample user interfaces created with McDE.

Figure 11: The MoDE is used to edit itself.

To demonstrate the generality of MoDE, the user interface of MoDE was created using itself. Consequently, MoDE can be used to edit itself. For example, in Figure 11, the user is using MoDE to examine the connection between the "ShrinkBox" and the "Window" of the interaction technique library. The user has also made some changes to MoDE. The two scroll bars of the interaction technique library were removed, and a "Roam Box" (a two-dimensional scrolling device) has been attached.

Since it is easy for users to customize the user interface of MoDE, the interface images shown in this document represent only a small sample of those developed by the author.

Rapid Prototyping

In an informal experiment to study the effectiveness of MoDE, two groups of subjects were asked to create the same interface. One group used MoDE exclusively; the other group used whatever tools they liked except MoDE. The group using MoDE were able to finish the assignment both faster and with fewer unimplemented features than the other group. Time data collected from this informal experiment suggest that MoDE reduces the time required

13

to develop a prototype interface by nearly an order of magnitude.

5 Conclusion

MoDE provides an effective environment for prototyping user interfaces. The capability to easily incorporate new objects into the library results in a general system with which a wide variety of interfaces can be created. Experience with MoDE shows that its support for connection between the user interface and the application substantially facilitates the prototyping process.

To support studies of user behaviors, ι experimental tracking system has been incorporated into MoDE to collect transcripts of users' interaction with prototype interfaces. The transcripts can be used to recreate the users' sessions and to support computerized analysis. MoDE is also being used to prototype interfaces for a hypertext software engineering system.

6 Acknowledgement

References

[Car89] L. Cardelli. Building user interface with direct manipulation.
SIGCHI'89: Human Factors in Computing Systems, pages 152–
166, May 1989.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its
Implementation.* Addison-Wesley, 1983.

[LIBY89] T. G. Lewis, Fred Handloser III, Sharada Bose, and Sherry
Yang. Prototypes from standard user inferface management sys-
tem. *Communications of the Assoication of Computing Machinery*,
22(5):51–60, may 1989.

[NeX88] NeXT, Inc., Palo Alto, Calif. *NeXT System Reference Manual*,
1988.

[Sha89] Yen-Ping Shan. An event-driven model-view-controller framework
for smalltalk. In *OOPSLA '89: Object Oriented Programming, Sys-
tems and Applications*, pages 347–352, October 1989.

[Sme87] SmethersBarnes, P.O. Box 639, Portland, Ore. 97207. *Smethers-
Barnes Prototyper User's Manual*, 1987.

15